A Formal Framework to Specify and Verify Real Time Properties on Critical Systems

Didier Le Botlan, Silvano Dal Zilio, Nouha Abid

A Formal Framework to Specify and Verify Real Time Properties on Critical Systems

Nouha Abid Silvano Dal Zilio Didier Le Botlan
CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse
Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France

February 10, 2014

Abstract

We propose a verified approach to the formal verification of timed properties using model-checking techniques. We focus on properties commonly found during the analysis of reactive systems, expressed using real-time specification patterns. We use observers in order to transform the verification of these timed patterns into the verification of simpler LTL formulas. While the use of observers for model-checking is quite common, our contribution is original in several ways. First, we define a formal framework to verify that observers are correct and non-intrusive. Second, we define different classes of observers for each pattern and use a pragmatic approach in order to select the most efficient candidate in practice. This approach is implemented in an integrated verification tool chain for the Fiacre language.

Keywords: Verification, Requirement, Specification, Patterns, Model Checking, Observers, Real Time Systems, Time Petri Nets, Formal Methods.

1 Introduction

An issue limiting the adoption of model checking technologies by the industry is the difficulty, for non-experts, to express their requirements using the specification languages supported by the verification tools. Indeed, there is often a significant gap between the boilerplates used in requirement statements and the low-level formalisms used by model checking tools; the latter usually relying on temporal logic. This limitation has motivated the definition of new languages, methods and tools to specify and verify properties. Hence, the following questions: "How to express the requirements of a real-time system?", "How to check real-time properties?", "How to verify that the results obtained after verification are correct (meaningful)?" or to rephrase it, "who checks that the model checker is correct?".

Many works have been done in this context, which we can divide into three main categories: works related (1) to the *specification* of real time properties; (2) to the *verification* of such properties; and (3) to the soundness proof of these verification approaches.

Regarding (1), two categories of works have been proposed to specify properties. The first category is based on temporal logics. It provides most of the theoretically well-founded body of works, such as complexity results for different fragments of real-time temporal logics Henzinger (1998): Temporal logic with clock constraints (TPTL); Metric Temporal Logic (MTL, MITL); Event Clock Logic; etc. The algebraic nature of logic-based approaches make them expressive and enable an accurate formal semantics. However, it may be impossible to express all the necessary requirements inside the same logic fragment if we ask for an efficient model-checking algorithm (with polynomial time complexity). For example, Uppaal Behrmann et al. (2004) choose a restricted fragment of TCTL with clock variables, while Kronos provides a more expressive framework, but at the cost of a much higher complexity. As a consequence, this approach requires different model-checkers for each interesting fragment of these logics—and a way to choose the right tool for every requirement—which may be impractical. The second category is based on specification patterns. Patterns propose a user-friendly syntax which facilitates their adoption by non-experts. However, in the real-time case, most of these approaches lack in theory or use inappropriate definitions. One of our goal is to reverse this situation. In the seminal work of Dwyer et al. (1999), patterns are defined by translation to formal frameworks, such as LTL and CTL. There is no need to provide a verification approach, in this case, since efficient model-checkers are available for these logics. This work on patterns has been extended to the real-time case. For example, Konrad & Cheng (2005) extends the patterns language with time constraints and give a mapping from timed pattern to TCTL and MTL, but they do not study the decidability of the verification method (the implementability of their approach). Another related work is Gruhn & Laue (2006), where the authors define observers based on Timed Automata for each pattern. However, they lack a formal framework for proving the correctness or the innocuousness of their observers and have not integrated their approach inside a model-checking tool chain.

Concerning verification by means of observers (2), Aceto et al. (2003, 1998) use test automata to check safety and bounded liveness properties of reactive systems. A similar approach has been experimented by Toussaint et al. (1997) on Time Petri Nets, but they propose a less general model for observers and consider only two verification techniques over four kinds of time constraints. Bayse et al. (2005) propose a formal method to verify the correctness of their approach, however, they do not prove all their invariants (patterns in our case).

Few works consider the verification of model-checking tools (3). Indeed, most of the existing approaches focus on the verification of the model-checking algorithm, rather than on the verification of the tool itself. For example, using the Isabelle theorem prover, Schimpf et al. (2009) introduce a proved algorithm for generating Bchi automata from a LTL formula. This algorithm is at the heart of many LTL model-checker based on an automata-theoretic approach. The issue of verifying verification tools also appears in conjunction with certification issues. In particular, many certification norms, such as the DO-178B, requires that any tool used for the development of a critical equipment be qualified at the same level of criticality than the equipment. (Of course, certification does not necessarily mean formal proof!) For instance, consider the certification of the SCADE compiler Esterel Technologies (n.d.), a tool-suite based on the synchronous language Lustre that integrates a model-checking engine. Nonetheless, only the code-generation part of the compiler is certified and not the verification part.

A main motivation for the work performed in this paper is to fill some of the gaps left by the existing solutions. Our goal is to develop a complete framework to specify and verify properties in the context of *real time critical systems*, that is, where the correctness of the system depends upon timing constraints, such as the "timeliness" of some interactions. For the implementation part of this study, we have worked with Fiacre, Berthomieu et al. (2008)—a formal language for the specification of real time systems developed in our team—and the Tina toolbox, Berthomieu et al. (2004), a model-checker for Timed Petri Nets and their extensions.

In this paper, we propose a complete framework that includes: the definition of timed patterns; an approach for checking timed properties; and methods for proving the correctness of this verification scheme. This article is an extension of the workshop article by Abid et al. (2012), providing more detailed material (use cases, patterns, observers), and simplifying the formal treatment thanks to a new definition of timed traces, as was suggested by former reviewers.

Contributions Our first contribution to the specification of requirements is the definition of real time patterns that can be viewed as a real time extension of patterns formerly defined by Dwyer et al. (1999). (A recent study by Bianculli et al. (2012) has shown that Dywer's patterns are the most used in practice in industry and academia.) As an alternative to timed extensions of temporal logic, we propose patterns designed to express general timing constraints commonly found in the analysis of real time systems (such as compliance to deadlines; event duration; bounds on the worst-case traversal time; etc.). They are also designed to be simple in terms of both clarity and computational complexity. In particular, each pattern should correspond to a decidable model-checking problem. We believe that this approach may ease the task of system engineers that are not well trained for the use of formal verification techniques.

When compared with temporal logics, our catalog of real time patterns is less expressive, but suffices to express most common requirements. Indeed, in the case of untimed patterns, Dwyer shows through a study of 500 specification examples that 80% of the temporal requirements can be covered by a small number of patterns. Moreover, we are able to implement an automatic verification method, using a less complex algorithm and with an overall better runtime and space complexity than with a model-checker for a "full-fledged" timed temporal logic.

Our second contribution is a verification method for checking timed patterns on real-time models. By grafting observers to these models, we simplify the verification of timed patterns into the verification of simple LTL formula. While the use of observers for model checking is quite common, our contribution is original in one way. Indeed, we define different classes of observers for each pattern and use a pragmatic approach in order to select the most efficient candidate in practice, in terms of verification time and system size growth.

Our third contribution is the definition of a formal framework to verify that our observers are sound and non-intrusive, meaning that they compute the correct answer and have no impact on the system under observation. The formal framework we have defined is not only useful for proving the validity of formal results but also for checking the soundness of optimisations in the implementation. We use this theoretical proof method for checking the correctness of our best "candidates".

Outline The formal framework used to prove the correctness of the verification approach is introduced in Section 2. Section 3 contains our catalogue of real time patterns, while Section 4 deals with our verification scheme. Section 5 details the observers we use to check our patterns. Before concluding, we present some use cases in Section 6.

2 Formal framework

This section introduces the formal foundations we rely on to provide a sound semantics to patterns, and to sustain our verification approach. We are based on an extension of Time Petri Nets (TPN), Merlin (1974), with shared variables and priorities, called Time Transition Systems (TTS) (Def. 2). The behaviour of a Time Transition System is abstracted as a set of traces, called Timed Traces (Def.3).

2.1 Time Transition Systems

Time Transition Systems (TTS) is an internal format used in our model-checking tool. It is the output of the Fiacre compiler, Berthomieu (2012).

Informal Definition We introduce TTS models thanks to a graphical syntax, as shown in Fig. 1. Ignoring at first side conditions and side effects (the *pre* and *act* expressions inside dotted rectangles), the TTS in Fig. 1 can be viewed as a TPN with one token in place p_0 as its initial marking. From this "state", a click transition may occur and move the token from p_0 to p_1. With this marking, the internal transition τ is enabled and will fire after exactly one unit of time, since the token in p_1 is not consumed by any other transition. Meanwhile, the transition labeled click may fire one or more times without removing the token from p_1, as indicated by the *read arc* (the one with a black dot). After exactly one unit of time, because of the priority arc (a dashed arrow between transitions), the click transition is disabled until the token moves from p_1 to p_2.

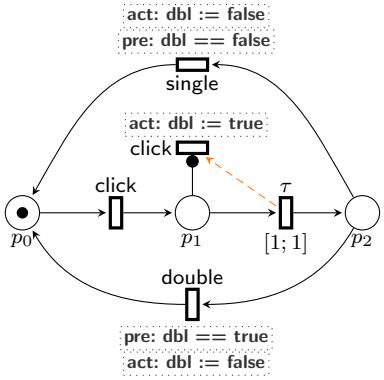


Figure 1: A double-click example in TTS

Data is managed within the *act* and *pre* expressions that may be associated to each transition. These expressions may refer to a fixed set of variables that form the *store* of the TTS. Assume t is a transition with guards act_t and pre_t. In comparison with a TPN, a transition t in a TTS is enabled if there is both: (1) enough tokens in the places of its pre-condition; and (2) the predicate pre_t is true. With respect to the firing of t, the main difference is that we modify the store by executing the action guard act_t. For example, when the token reaches the place p_2 in the TTS of Fig. 1, we use the value of the variable dbl to test whether we should signal a double click or not.

Formal Definition Since the TTS model is an extension of TPN, we first present a formal definition of the latter. Labeled Time Petri Nets (or TPN) extends Time Petri Nets, Merlin (1974), with an action alphabet and a function labelling the transitions with those actions.

Notation : Let I^+ be the set of nonempty real intervals with non negative rational endpoints. For $i \in I^+$, the symbol $\downarrow i$ denotes the left end-point of the interval i and $\uparrow i$ its right end-point, if i is bounded, or ∞ otherwise. We use $\mathbb{N}$ to denote the set of non negative integers.

Definition 1 *A labeled Time Petri Net (or TPN) is a 8-tuple* $(P, T, B, F, M^0, I^s, \mathcal{L}, L)$*, in which:*

- P *is a finite set of places* p_i *;*
- T *is a finite set of transitions* t_i *;*
- B *is the backward incidence function* $\quad B : T \to P \to \mathbb{N}$ *;*
- F *is the forward incidence function* $\quad F : T \to P \to \mathbb{N}$ *;*
- M^0 *is the initial marking function* $\quad M^0 : P \to \mathbb{N}$ *;*
- I^s *is a function called the static interval function* $\quad I^s : T \to I^+$

  I^s *associates a temporal interval* $I^s(t) \in I^+$ *with every transition of the system. Assuming that a transition* t *became enabled at time* τ*, then* t *cannot fire before* $(\tau + \downarrow I^s(t))$ *and no later than* $(\tau + \uparrow I^s(t))$ *unless disabled by firing some other transition.*
- $\mathcal{L}$ *is a finite set of actions, or labels, not containing the silent action* ε*;*
- $L : T \to \mathcal{L} \cup \{\varepsilon\}$ *is a transition labelling function.*

A marking *is a function* $M : P \to \mathbb{N}$ *that records the current (dynamic) value of the places in the net, as transitions are fired. The transition* $t \in T$ *is enabled by* M *iff* $M \geqslant B(t)$*. The* dynamic interval function $I : T \to I^+$ *is a mapping from transitions to time intervals. It is used to record the current timing constraints associated to each transition, as time passes. A* state *of the TPN is a pair* (M, I)*. Its* initial state *is* (M^0, I^s)*.*

In the state (M, I), a transition t can fire iff t is enabled at M and instantly fireable, that is $0 \in I(t)$. In the target state, the transitions that remained enabled while t is fired (t excluded) keep their time interval, the intervals of the others (newly enabled) transitions are set to their respective static intervals. Together with those "discrete" transitions, a Time Petri Nets adds the ability to model the flowing of time. A continuous transition of amount d (i.e. taking d time units) is possible iff d is less than $\uparrow I(t)$ for all enabled transitions t.

The definition of TTS is a natural extension of TPN that takes variables and priorities into account. We present, next, a definition of TTS based on Definition. 1.

The main difference lies, on the one hand, in the priority relation on transitions, and, on the other hand, in the fact that each transition has preconditions and postconditions, interpreted relatively to a *store*.

Definition 2 *(Time Transition Systems) A Time Transition Systems (or TTS) is a 10-tuple* $(P, T, B, F, M^0, I^s, \mathcal{L}, L, S, <)$ *where:*

- $(P, T, B, F, M^0, I^s, \mathcal{L}, L)$ *is a labeled Time Petri Net.*
- S *is a set of stores* s^i. s^0 *is the initial store. To each store* s*, we associate a partial function* $f_s : T \to S$*, whose domain contains only the transitions whose preconditions are satisfied by* s*, and such that* $f_s(t)$ *is the store* s *updated by post-conditions of* t*.*
- $<$ *is a binary, transitive relation over* T *which encodes the (static) priority relation between transitions;*

The transition $t \in T$ *is* enabled *by a marking* M *and a store* $s \in S$ *iff* $M \geqslant B(t)$ *and* $f_s(t)$ *is defined. A transition is* time-enabled *if it is enabled and* $0 \in I(t)$*. It is* fireable *if it is time-enabled and there is no time-enabled transition* t' *that has priority over* t *(that is* $t < t'$*). In the target state, time intervals and markings are updated as in Time Petri Nets, and the new store is* $f_s(t)$*.*

The *state* of a TTS is a triple (M, s, I). Its initial state is (M^0, s^0, I^s).

2.2 Semantics of Time Transition Systems expressed as Timed Traces

The execution of a TTS results in a set of timed sequences of events called *Timed Traces*. Formally, we define a timed event e as a pair (t, τ) recording the transition t and the date τ at which it was fired. We denote $\mathcal{E}$ the set $T \times \mathbb{R}^+$ of events. For convenience, we write d the second projection: $d : \mathcal{E} \to \mathbb{R}^+$ is such that $d(t, \tau) = \tau$.

Definition 3 (Timed Trace) *A Timed Trace* σ *is a (possibly infinite) sequence of timed events* $e \in \mathcal{E}$*. Formally,* σ *is a partial mapping from* $\mathbb{N}$ *to* $\mathcal{E}$ *such that* $\sigma(i)$ *is defined whenever* $\sigma(j)$ *is defined and* $i \leqslant j$*. The domain of* σ *is written* $\mathsf{dom}\,\sigma$*. Additionally, dates are required to be weakly increasing:* $\forall i, j \in \mathsf{dom}\,\sigma, i \leqslant j \Rightarrow d \circ \sigma(i) \leqslant d \circ \sigma(j)$*.*

We write $d(\sigma)$ the set $\{d(e) \mid e \in \mathsf{codom}\,\sigma\}$. Its least upper bound and greatest lower bound are written $\mathsf{sup}(d(\sigma))$ and $\mathsf{inf}(d(\sigma))$, respectively. They indicate the temporal range of σ. Using classic notations for sequences, the empty sequence is denoted ϵ; given a finite sequence σ and a—possibly infinite—sequence σ', we denote $\sigma.\sigma'$ the *concatenation* of σ and σ', provided $\mathsf{sup}(d(\sigma)) \leqslant \mathsf{inf}(d(\sigma'))$. We say that σ is a *prefix* of $\sigma.\sigma'$. The concatenation operation is associative.

Infinite traces are expected to have an infinite duration. Indeed, to rule out Zeno behaviours, which means that the system includes an infinite number of discrete steps in a finite amount of time, we only consider traces that let time elapse. Hence, the following definition:

Definition 4 (Well-formed Traces) *A trace* σ *is* well-formed *if and only if its domain is finite or* $\mathsf{sup}(d(\sigma)) = +\infty$

We capture the semantics of TTS as a labeled relation between states, $(M, s, I) \longmapsto^l (M', s', I')$, where l is either a delay $\delta \in \mathbb{R}^+$ or a transition $t \in T$. Like a TPN, a TTS with state (M, s, I) may progress in two ways:

- *Time elapses* by an amount δ in I^+, provided $\delta \in I(t)$ for all enabled transitions, meaning that no transition t is urgent. In that case, we define $I'(t) = I(t) - \delta$ for

all enabled transitions t and $I'(t) = I^s(t)$ for disabled transitions. Under these hypotheses, we have

$$(M, s, I) \longmapsto^{\delta} (M, s, I')$$

- *A fireable transition t fires*:

$$(M, s, I) \longmapsto^{t} (M', s', I')$$

such that M', s' and I' are respectively the new marking, store and dynamic interval function, obtained after firing t.

Given a sequence of reductions $st_0 \longmapsto^{l_1} st_1 \longmapsto^{l_2} st_2 \ldots \longmapsto^{l_n} st_n$, we can build a timed trace as described next.

Definition 5 *We define the ternary relation $st \longrightarrow^{\sigma} st'$ over pairs of states and finite timed traces as the smallest relation satisfying the following inference rules:*

$$\frac{}{st \longrightarrow^{\epsilon} st} \qquad \frac{st \longmapsto^{t} st' \quad \tau \in \mathbb{R}^+}{st \longrightarrow^{(t,\tau)} st'} \qquad \frac{st_1 \longrightarrow^{\sigma} st_2 \quad st_2 \longmapsto^{\delta} st_3 \quad st_3 \longmapsto^{t} st_4 \quad \tau' = \mathsf{sup}(d(\sigma)) + \delta}{st_1 \longrightarrow^{\sigma.(t,\tau')} st_4}$$

We write $st \longrightarrow^{\sigma}$ whenever there exist a state st' such that $st \longrightarrow^{\sigma} st'$. Given an infinite trace σ, we write $st \longrightarrow^{\sigma}$ if and only if $st \longrightarrow^{\sigma'}$ holds for all σ' finite prefixes of σ. Finally, the set of traces of a TTS N is the set of well-formed traces σ such that $(M^0, s^0, I^s) \longrightarrow^{\sigma}$ holds. This set is written $\Sigma(N)$.

2.3 Composition of Time Transition Systems and Composition of Timed Traces

We study the composition of two TTS and consider the relation between traces of the composed system and traces of both components. This operation is particularly significant in the context of this work, since both the system and the observer are TTS and we use composition to graft the latter to the former. In particular, we are interested in conditions ensuring that the behaviour of the observer does not interfere with the behaviour of the observed system.

The "parallel composition" of Labeled Time Petri Nets is a fundamental operation that is used to model large systems by incrementally combining smaller nets. Basically, the composition of two Labeled TPN N_1 and N_2 is a Labeled net $N \stackrel{\text{def}}{=} (N_1 \parallel N_2)$ such that: the places of N is the cartesian product of the places of N_1 and N_2, and the transitions of N is the fusion of the transitions in N_1 and N_2 that have the same label. A formal definition for the composition of two TPN is given in Peres et al. (2011).

Composition of TTS is basically the same, with the noticeable restriction that transitions which have priority over other transitions may not be synchronised across components. This is required to ensure the composition property 1.

Definition 6 (Composable TTS, synchronised transitions) *We consider two TTS, namely N_1 and N_2, defined as $(P_i, T_i, \ldots)$ for $i \in \{1, 2\}$, respectively. The set of* synchronised transitions *of N_1 is $\{t_1 \in T_1 \mid \exists t_2 \in T_2 \;.\; L_1(t_1) = L_2(t_2)\}$. We define the set of synchronised transitions of N_2 similarly. Then, N_1 and N_2 are* composable *iff the following conditions hold:*

1. *$P_1 \cup T_1 \cup S_1$ is disjoint from $P_2 \cup T_2 \cup S_2$.*
2. *for $i = 1, 2$, every synchronised transition t_i of N_i is such that $I_i^s(t_i) = [0, +\infty[$, and there is no transition $t' \in T_i$ with $t' <_i t_i$.*

The first condition ensures that N_1 and N_2 are disjoint. As stated by the second condition, in every pair of synchronised transitions, both transitions must have a trivial time constraint $[0, +\infty[$. This condition as well as the condition on priorities is necessary to ensure composition, as stated by Property 1 below.

Definition 7 (Composition of two TTS) *Assuming N_1 and N_2 are defined as above, let N be the TTS corresponding to their composition, which we write $N = N_1 \parallel N_2$. It is defined by a 10-tuple $(P, T, B, F, M^0, I^s, \mathcal{L}, L, S, <)$ where, in particular:*

1. *$P = P_1 \cup P_2$, $S = S_1 \times S_2$, $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$*
2. *Let $\perp$ be an element not in $T_1 \cup T_2$. Let $T_1^\perp$ be $T_1 \cup \{\perp\}$ and $T_2^\perp$ be $T_2 \cup \{\perp\}$. We define T as the following subset of $T_1^\perp \times T_2^\perp$:*
$$\begin{aligned} T \quad = \quad & \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, L_1(t_1) = L_2(t_2)\} \\ \cup \quad & \{(t_1, \perp) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronised}\} \\ \cup \quad & \{(\perp, t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronised}\} \end{aligned}$$
3. *Other elements (markings, incidence functions, static intervals) are, intuitively, obtained by merging elements of N_1 and N_2. By lack of space, we omit the details. The full definition is available in the appendix.*

We now define composition of timed traces, and then show, in Property 1, that traces generated by N correspond to composition of traces from N_1 and traces from N_2.

We say that an event e is synchronised iff e is (t, τ) and t is synchronised (as defined in Def. 6). In the same way that systems can be composed, it is possible to compose a timed trace of a TTS N_1 with the trace of another TTS N_2 when some conditions are met. Basically, events with the same label must occur synchronously, and unsynchronised events are freely interleaved.

Definition 8 (Composition of finite traces) *Assume N_1, N_2, and N are defined as in Def 7. We define a ternary relation $\sigma_1 \bowtie \sigma_2 \sim \sigma$ between σ_1, σ_2, and σ, finite timed traces of N_1, N_2, and N (respectively). The relation means that σ is an acceptable composition of σ_1 and σ_2. It is defined as the smallest relation satisfying the following inference rules:*

$$\epsilon \bowtie \epsilon \sim \epsilon$$

$$\textsc{Sync}\ \frac{L_1(t_1) = L_2(t_2) \qquad \sigma_1' \bowtie \sigma_2' \sim \sigma'}{(t_1, \tau)\sigma_1' \bowtie (t_2, \tau)\sigma_2' \sim ((t_1, t_2), \tau)\sigma'}$$

$$\textsc{Unsync-left}\ \frac{t_1 \text{ not synchronised} \qquad \tau \leqslant \mathsf{inf}(d(s_2)) \qquad \sigma_1' \bowtie \sigma_2 \sim \sigma'}{(t_1, \tau)\sigma_1' \bowtie \sigma_2 \sim ((t_1, \perp), \tau)\sigma'}$$

$$\textsc{Unsync-right}\ \textit{(similar)}$$

Rule Sync states that synchronised transitions must occur at the same time in both traces σ_1 and σ_2. Rule Unsync-left (resp. Unsync-right) states that an unsynchronised transition t_1 (resp. t_2) must appear as $(t_1, \perp)$ (resp. $(\perp, t_2)$) in the composed trace σ.

Definition 9 (Composition of traces) *Given three (possibly infinite) traces σ_1, σ_2, and σ, the relation $\sigma_1 \bowtie \sigma_2 \sim \sigma$ holds if and only if for all date $\tau \in \mathbb{R}^+$, there exist finite prefixes σ'_1, σ'_2 and σ', of σ_1, σ_2, and σ (respectively), such that $\tau \leqslant \mathsf{sup}(d(\sigma_1)) \Rightarrow \tau \leqslant \mathsf{sup}(d(\sigma'_1))$, $\tau \leqslant \mathsf{sup}(d(\sigma_2)) \Rightarrow \tau \leqslant \mathsf{sup}(d(\sigma'_2))$, and $\sigma'_1 \bowtie \sigma'_2 \sim \sigma'$ holds. We write $\sigma_1 \bowtie \sigma_2$, whenever there exists σ such that $\sigma_1 \bowtie \sigma_2 \sim \sigma$ holds.*

We are now able to state the composition property. Remind that $\Sigma(N)$ is the set of traces of N.

Property 1 (Compositionality) *Assume N_1 and N_2 are composable systems with events in $\mathcal{E}_1$ and $\mathcal{E}_2$ respectively. Let N be $N_1 \parallel N_2$; we write $\mathcal{E}$ its set of events. Then, both propositions hold:*

$$\forall \sigma \in \Sigma(N), \exists \sigma_1 \in \Sigma(N_1), \sigma_2 \in \Sigma(N_2) \ . \ \sigma_1 \bowtie \sigma_2 \sim \sigma$$

$$\forall \sigma_1 \in \Sigma(N_1), \sigma_2 \in \Sigma(N_2), \sigma \in \mathcal{E}^* \ . \ \sigma_1 \bowtie \sigma_2 \sim \sigma \Rightarrow \sigma \in \Sigma(N)$$

In other words, given a trace $\sigma \in \Sigma(N)$, one may extract two composable traces $\sigma_1 \in \Sigma(N_1)$ and $\sigma_2 \in \Sigma(N_2)$. Conversely, given two composable traces $\sigma_1 \in \Sigma(N_1)$ and $\sigma_2 \in \Sigma(N_2)$, their composition is guaranteed to be in $\Sigma(N)$. Thus, this property characterises the set of traces of N in terms of traces of N_1 and N_2.

By lack of space, we only provide a proof sketch. Given $\sigma \in \Sigma(N)$, we build σ_1 by mapping $(t_1, t_2) \in T$ to t_1 and by discarding events of the form $((\perp, t_2), \tau)$. One has to show, then, that the resulting trace belongs to $\Sigma(N_1)$. This is done by induction on the size of the considered finite prefix (see Def. 5 and related notations). The trace σ_2 is built similarly. It is easy, then, to check that $\sigma_1 \bowtie \sigma_2 \sim \sigma$ holds. Conversely, assume $\sigma_1 \bowtie \sigma_2 \sim \sigma$ holds for some traces $\sigma_1 \in \Sigma(N_1)$, $\sigma_2 \in \Sigma(N_2)$, and $\sigma \in \mathcal{E}^*$. Then, thanks to Definitions 7 and 8, σ can be proven to be a trace of N by induction on the size of finite prefixes of σ_1, σ_2, and σ.

This result is used to show the innocuousness of observers in Section 4.3.

2.4 Formal Framework for Expressing Timed Properties

We will use different methods to define the semantics of patterns (that is, essentially, to define the set of timed traces for which the pattern holds). Our experience shows that being able to confront different definitions for the same pattern, using contrasting approaches, is useful for teaching patterns.

2.4.1 Metric Temporal Logic

Metric Temporal Logic (MTL), Ouaknine & Worrell (2007); Koymans (1990), is an extension of LTL where temporal modalities can be constrained by a time interval. For instance, the MTL formula $A \ \mathbf{U}_{[1,3[} \ B$ states that in every trace of the system, the event B must occur at a time $t_0 \in [1, 3[$ and that A holds everywhere in the interval $[0, t_0[$. A MTL formula is defined as follows:

$$\phi ::= p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \bigcirc_I \phi \mid \phi_1 \ U_I \ \phi_2 \mid \Diamond_I \phi \mid \Box_I \phi$$

where p is a proposition, $I \subseteq \mathbb{R}^+$ is an open, closed, or half-open interval with end points in $\mathbb{N} \cup \infty$, $\bigcirc$ represents the *next* operator, U_I represents the *until* operator and $\Diamond$ and $\Box$ represent the *eventually* and *globally* operator respectively. We will use also a weak version of the "until modality", denoted $A\ W\ B$, that does not require B to eventually occur.

An advantage of using MTL is that it provides a sound and unambiguous framework for defining the meaning of patterns. On the negative side, however, it is known that the model-checking problem for full MTL is undecidable. Some works, see e.g. Ouaknine & Worrell (2007), have been done to find suitable decidable fragments of MTL. One such subset, called MITL, is obtained by disallowing punctual time intervals in formulas, that is, intervals of the form $[d, d]$.

2.4.2 First Order Formula over Timed Traces

Using MTL to define our patterns partially defeats one of the original goal of patterns, that is to circumvent the use of temporal logic in the first place. For this reason, we propose an alternative way for defining the semantics of patterns that relies on first-order formulas over timed traces (FOTT). Timed traces have been defined in Def. 3.

Assume A is a predicate overs transitions. By abuse of notation, we write (A, τ) to denote an event (t, τ) such that t satisfies A. Given a timed trace σ, we write $A \in \sigma$ for the formula $\exists \sigma_1, \sigma_2, \tau.\, (\sigma = \sigma_1.(A, \tau).\sigma_2)$. Similarly, if A is a state predicate, then the notation $\sigma(A)$ means that A holds for all the intermediate states visited through σ.

We can give more useful examples of FOTT formulas that corresponds to the notion of "scope" found in Dwyer's patterns. We consider a timed trace σ and predicates A and B. The "part" (or sub-trace) of σ occurring after the first occurrence of A—or simply σ **after** A—can be defined as the trace σ_2 such that:

$$\exists \sigma_1, \tau\ .\ \sigma = \sigma_1.(A, \tau).\sigma_2 \wedge A \notin \sigma_1$$

Likewise, the scope σ **before** A—which determines the part of σ located before the first occurrence of A—can be defined as the trace σ_1 such that:

$$\exists \sigma_2, \tau\ .\ \sigma = \sigma_1.(A, \tau).\sigma_2 \wedge A \notin \sigma_1$$

Finally, we can define constraints such as: the duration "**between** A and B" is equal to d—which determines the trace σ located between the first occurrence of A and the first occurrence of B—using the FOTT formula:

$$\exists \sigma_1, \tau_a, \tau_b\ .\ \sigma = (A, \tau_a).\sigma_1.(B, \tau_b) \wedge \tau_b - \tau_a = d$$

We believe that the use of FOTT may ease the work of engineers who are not trained with formal verification techniques but who have a background on mathematical analysis.

3 Real-time specification patterns

A pattern is a template that specifies a property which occurs commonly in the specification of concurrent and reactive systems; We describe, in this section, our patterns

using a hierarchical classification borrowed from Dwyer Dwyer et al. (1999) but adding the notion of "timing modifiers". Our patterns are built from five categories, listed below, or from the composition of several patterns:

- **Existence Patterns (Present):** conditions that must eventually occur;
- **Absence Patterns (Absent):** conditions that should not occur;
- **Universality Patterns:** conditions that must occur throughout the whole execution;
- **Response Patterns (Response):** (trigger) conditions that must always be followed by a given (response) condition;
- **Precedence Patterns :** (signal) conditions that must always be preceded by a given (trigger) condition.

In each class, generic patterns may be specialised using one of five *scope modifiers* that limit the range of the execution trace over which the pattern must hold:

- **Global :** the default scope modifier, that does not limit the range of the pattern. The pattern must hold over the whole timed trace;
- **Before R :** limit the pattern to the beginning of a timed trace, up to the first occurrence of R;
- **After Q :** limit the pattern to the events following the first Q;
- **Between Q and R :** limit the pattern to the events occurring between an event Q and the following occurrence of an event R;
- **After Q Until R :** similar to the previous scope modifier, except that we do not require that R must necessarily occur after a Q.

Finally, timed patterns are obtained using one of four possible kinds of *timing modifiers* that limit the possible dates of events referred in the pattern:

- **Within I, For interval I :** to constraint the delay between two given events to belong to the time interval I;
- **Lasting D, For duration D :** to constraint the length of time during which a given condition holds (without interruption) to be greater than D.

For each pattern, we present a textual description and an example inspired from the TTS example in Fig. 2. This TTS models an airlock containing two doors (D_1 and D_2) and two buttons. At any time, at most one door can be open.

Our model includes two boolean variables, req_1 and req_2, indicating whether a request to open door D_1 (resp. D_2) is pending. Those variables are read by pre-conditions on transitions Open_i, Button_i, and $\mathsf{Shutdown}$ and are modified by post-actions on transitions Button_i and Close_i. For instance, the pre-condition $\neg\mathsf{req}_2$ on Button_2 is used to disable the transition when the door is already open. This implies that pressing the button while the door is open has no further effect.

Moreover, for each pattern, we provide both its denotational interpretation based on FOTT, and a logical definition based on MTL. Next, we use the symbol I as a shorthand for the time interval $[d_1, d_2]$.

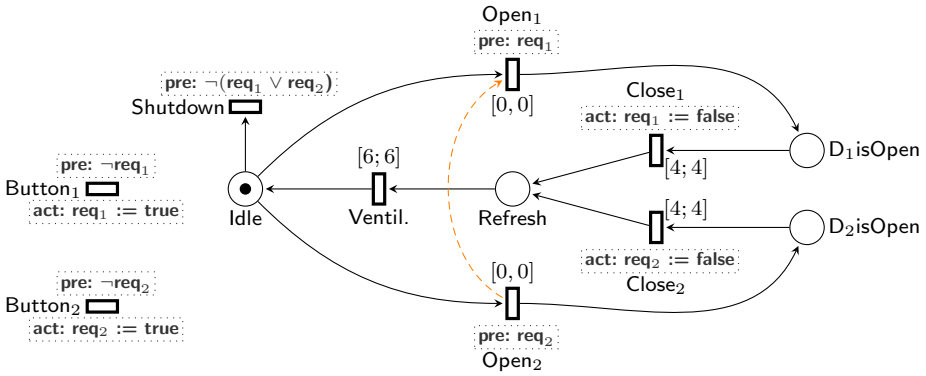


Figure 2: The Airlock system: at any time, at most one door is open. When a door closes, some ventilation takes place for 6 u.t.

3.1 Existence patterns

An existence pattern is used to express that, in every trace of the system, some events must occur.

Present A **after** B **within** I

Predicate A must hold between d_1 and d_2 units of time (u.t) after the first occurrence of B. The pattern is also satisfied if B never holds.

Example: **present Ventil. after** $(\textbf{Open}_1 \vee \textbf{Open}_2)$ **within** $[0, 10]$

MTL def.: $(\neg B)\ \mathbf{W}\ (B \wedge \mathit{True}\ \mathbf{U}_I\ A)$

FOTT def.: $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1(B, \tau_1)\sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4\ .\ \sigma_2 = \sigma_3(A, \tau_2)\sigma_4 \wedge \tau_2 - \tau_1 \in I$

Present first A **before** B **within** I

The first occurrence of predicate A holds between d_1 and d_2 u.t. before the first occurrence of B. The pattern is also satisfied if B never holds. (The difference with **Present** B **after** A **within** I *is that B should not occur before the first A.)*

Example: **present first** $\textbf{Button}_1 \vee \textbf{Button}_2$ **before** $(\textbf{Open}_1 \vee \textbf{Open}_2)$ **within** $[0, 0]$

MTL def.: $(\Diamond B) \Rightarrow ((\neg A \wedge \neg B)\ \mathbf{U}\ (A \wedge \neg B \wedge (\neg B\ \mathbf{U}_I\ B)))$

FOTT def.: $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1(B, \tau_1)\sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4\ .\ \sigma_1 = \sigma_3(A, \tau_2)\sigma_4 \wedge A \notin \sigma_3 \wedge \tau_1 - \tau_2 \in I$

Present A **lasting** D

Starting from the first occurrence of predicate A, it remains true for at least duration D. The pattern makes sense only if A is a predicate on states (that is, on the marking or store); since transitions are instantaneous, they have no duration.

Example: **present Refresh lasting** 6

MTL def.: $(\neg A)\ \mathbf{U}\ (\Box_{[0,D]} A)$

FOTT def.: $\exists \sigma_1, \sigma_2, \sigma_3\ .\ \sigma = \sigma_1\sigma_2\sigma_3 \wedge A \notin \sigma_1 \wedge A(\sigma_2) \wedge \mathsf{inf}(d(\sigma_3)) - \mathsf{inf}(d(\sigma_2)) \geqslant D$

3.2 Absence patterns

Absence patterns are used to express that some condition should never occur.

Absent A **after** B **for interval** I

Predicate A must never hold between d_1–d_2 u.t. after the first occurrence of B. This pattern is dual to Present A after B within I (it is not equivalent to its negation because, in both patterns, B is not required to occur).

Example: **absent** $\mathbf{Open}_1 \vee \mathbf{Open}_2$ **after** $\mathbf{Close}_1 \vee \mathbf{Close}_2$ **for interval** $[0, 10]$

MTL def.: $\neg B \ \mathbf{W} \ (B \wedge \Box_I \neg A)$

FOTT def.: $\forall \sigma_1, \sigma_2, \sigma_3 \ . \ (\sigma = \sigma_1(B, \tau_1)\sigma_2\sigma_3 \wedge B \notin \sigma_1 \wedge \mathsf{inf}(d(\sigma_3)) - \tau_1 \in I) \Rightarrow A \notin \sigma_2$

Absent A **before** B **for duration** D

No A can occur less than D u.t. before the first occurrence of B. The pattern holds if there are no occurrence of B.

Example: **absent** $\mathbf{Open}_1$ **before** $\mathbf{Close}_1$ **for duration** 3

MTL def.: $\Diamond B \Rightarrow (A \Rightarrow (\Box_{[0,D]} \neg B)) \ \mathbf{U} \ B$

FOTT def.: $\forall \sigma_1, \sigma_2, \sigma_3 \ . \ (\sigma = \sigma_1\sigma_2(B, \tau_1)\sigma_3 \wedge B \notin \sigma_1\sigma_2 \wedge \tau_1 - \mathsf{inf}(d(\sigma_2)) \leqslant D) \Rightarrow A \notin \sigma_2$

3.3 Response patterns

Response patterns are used to express "cause–effect" relationship.

A **leadsto first** B **within** I

Every occurrence of A must be followed by an occurrence of B within time interval I (considering only the first occurrence of B after A).

Example: $\mathbf{Button}_2$ **leadsto first** $\mathbf{Open}_2$ **within** $[0, 10]$

MTL def.: $\Box(A \Rightarrow (\neg B) \ \mathbf{U}_I \ B)$

FOTT def.: $\forall \sigma_1, \sigma_2 \ . \ (\sigma = \sigma_1(A, \tau_1)\sigma_2) \Rightarrow \exists \sigma_3, \sigma_4 \ . \ \sigma_2 = \sigma_3(B, \tau_2)\sigma_4 \wedge \tau_2 - \tau_1 \in I \wedge B \notin \sigma_3$

A **leadsto first** B **within** I **before** R

Before the first occurrence of R, each occurrence of A is followed by a B—and these two events occur before R—in the time interval I. The pattern holds if R never occur.

Example: $\mathbf{Button}_2$ **leadsto first** $\mathbf{Open}_2$ **within** $[0, 10]$ **before Shutdown**

MTL def.: $\Diamond R \Rightarrow (\Box(A \wedge \neg R \Rightarrow (\neg B \wedge \neg R) \ \mathbf{U}_I \ B \wedge \neg R) \ \mathbf{U} \ R$

FOTT def.: $\forall \sigma_1, \sigma_2, \sigma_3 \ . \ (\sigma = \sigma_1(A, \tau_1)\sigma_2(R, \tau)\sigma_3 \wedge R \notin \sigma_1(A, \tau_1)\sigma_2 \Rightarrow \exists \sigma_4, \sigma_5 \ . \ \sigma_2 = \sigma_4(B, \tau_2)\sigma_5 \wedge \tau_2 - \tau_1 \in I \wedge B \notin \sigma_4$

A **leadsto first** B **within** I **after** R

*Like A **leadsto first** B **within** I but only considering occurrences of A after the first R.*

Example: $\mathbf{Button}_2$ **leadsto first** $\mathbf{Open}_2$ **within** $[0, 10]$ **after Shutdown**

MTL def.: $\Box(R \Rightarrow (\Box(A \Rightarrow (\neg B) \ \mathbf{U}_I \ B)))$

FOTT def.: $\forall \sigma_1, \sigma_2 \ . \ (\sigma = \sigma_1(R, \tau)\sigma_2(A, \tau_1)\sigma_3 \wedge R \notin \sigma_1) \Rightarrow \exists \sigma_4, \sigma_5 \ . \ \sigma_3 = \sigma_4(B, \tau_2)\sigma_5 \wedge \tau_2 - \tau_1 \in I \wedge B \notin \sigma_4$

3.4 Pattern Composition

Finally, patterns can be easily combined together using the usual boolean connectives (or, and, imply). The pattern P1 and P2 holds for all the traces where P1 and P2 both hold. The pattern P1 or P2 holds for all the traces where P1 or P2 holds. The pattern not P1 holds for all the traces where P1 does not hold.

4 Verification

We use timed observers to verify our timed patterns. Different types of observers are defined at the TTS level. Note that we do not provide an automatic method to generate observers. Rather, we define a set of observers for each patterns and, after selecting the "most efficient one", we prove that it is correct (see the discussion in Section. 4.3). We make use of the whole expressiveness of the TTS model to build observers: synchronous or asynchronous rendez-vous (through places and transitions); shared memory (through data variables); and priorities. We believe that an automatic method for generating the observer, while doable, will be detrimental for the performance of our approach. Moreover, when compared to a "temporal logic" approach, we are in a more favourable situation because we only have to deal with a finite number of patterns.

4.1 Observers for the leadsto Pattern

We focus on the example of the **leadsto** pattern. We assume that some events of the system are labeled with E_1 and some others with E_2. In our context, the event of a model can be: a transition that is fired, the system entering or leaving a state, a change in the value of variables, ... We give three examples of observers for the pattern: E_1 **leadsto** E_2 **within** $[0, max[$. The first observer monitors transitions and uses a single place; the second one monitors shared, boolean variables injected into the system (by means of composition); the third one monitors places. We define our TTS observers using a classical graphical notation for Petri Nets, where arcs with a black circle denote *read arcs*, while arcs with a white circle are *inhibitor arcs*. (These extra categories of arcs can be defined in TTS and are supported in our tool chain.) The use of a *data observer* is quite new in the context of TTS systems. The results of our experiments seem to show that, in practice, this is the best choice to implement an observer.

4.1.1 Transition Observer

The observer O_t, see Fig. 3, uses a place, obs, to record the time since the last transition E_1 occurred. The place obs in O_t is emptied if a transition labeled E_2 is fired, otherwise the transition error is fired after max unit of time. The priority arc (dashed arrow) between error and E_2 is used to observe the transition error even in the case where a transition E_2 occurs exactly max u.t. after the place obs was filled.

By definition of the TTS composition operator, the composition of the observer O_t with the system N duplicates each transitions in N that is labeled E_1: one copy can fire if obs is empty—as a result of the inhibitor arc—while the other can fire only if the place is full. As a consequence, in the TTS $N \parallel O_t$, the transition error can fire if and only if the place obs stays full—there has been an instance of E_1 but not of E_2—for

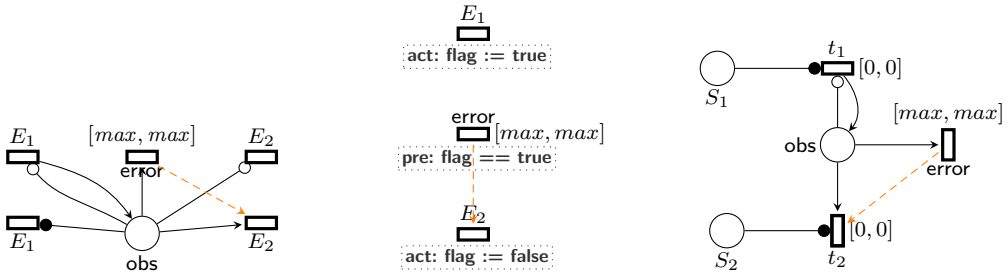


Figure 3: From left to right: transition observer O_t, data observer O_d, place observer O_p

a duration of max. Then, to prove that N satisfies the **leadsto** pattern, it is enough to check that the system $N \parallel O_t$ cannot fire the transition error. This can be done by checking the LTL formula $\Box(\neg\textbf{error})$ on the system $N \parallel O_t$.

The observer O_t given in Fig. 3 is *deterministic* and will observe every occurrence of E_1 in a given execution. It is also possible to define a non-deterministic observer, such that some occurrences of E_1 may be disregarded. This approach is safe since model-checking performs an exhaustive exploration of the states of the system; it considers all possible scenarios. This non-deterministic behaviour is quite close to the treatment obtained when compiling an (untimed) LTL formula "equivalent" to the **leadsto** pattern, namely $\Box(E_1 \Rightarrow \Diamond E_2)$, into a Bchi automaton Gastin & Oddoux (2001). We have implemented the deterministic and non-deterministic observers and compared them taking into account their impact on the size of the state graphs that need to be generated and on the verification time. Experiments have shown that the deterministic observer is more efficient, which underlines the benefit of singling out the best possible observer and looking for specific optimisation.

4.1.2 Data Observer

The data observer O_d, depicted in Fig. 3, features a transition error conditioned by the value of a boolean variable, flag, that "takes the role" of the place obs in O_t (every boolean variable is considered to be initially set to false). Indeed, flag is true between an occurrence of E_1 and the following transition E_2. Therefore, like in the previous case, to check if a system N satisfies the pattern, it is enough to check the reachability of the event error. Notice that the whole state of the data observer is encoded in its store, since the underlying net has no place.

4.1.3 Place Observer

The place observer O_p is also depicted in Fig. 3. In this section, to simplify the presentation, we assume that the events E_1 and E_2 are associated to the system entering some given states S_1 and S_2. (But we can easily adapt this net to observe events associated to transitions in the system.) We also rely on a composition operator that composes TTS through their places instead of their transitions Peres et al. (2011) and that is

available in our tool chain. In O_p, we use a transition t_1 whenever a token is placed in S_1 and a transition t_2 for observing that the system is in state S_2 (we assume that the labels t_1 and t_2 are fresh—private to the observer—and should not be composed with the observed system). The remaining component of O_p is just like the transition observer. We consider both a place and a transition observer since, depending on the kind of events that are monitored, one variant may be more efficient than the other.

4.2 Choice of the best observer in practice

Our verification framework has been integrated into a prototype extension of *frac*, the Fiacre compiler for the TINA toolbox.

We define the *empirical complexity* of an observer as its impact on the augmentation of the state space size with respect to the observed system. For a system S, we define $size(S)$ as the size (in bytes) of the *State Class Graph* (SCG), Berthomieu et al. (2004), of S generated by our verification tools. In TINA, we use SCG as an abstraction of the state space of a TTS. State class graphs exhibit good properties: an SCG preserves the set of discrete traces—and therefore preserves the validation of LTL properties—and the SCG of S is finite if the Petri Nets associated with S is bounded and if the set of values generated from S is finite. We cannot use the "plain" labeled transition system associated to S to define the size of S; indeed, this transition graph is usually infinite since we work within a dense time model.

The size of S is a good indicator of the memory footprint and the computation time needed for model-checking the system S: the time and space complexity of the model-checking problem is proportional to $size(S)$. Building on this definition, we say that the complexity of an observer O applied to the system S, denoted $C_O(S)$, is the quotient between the size of $(S \parallel O)$ and the size of S.

We resort to an empirical measure for the complexity since we cannot give an analytical definition of C_O outside of the simplest cases. However, we can give some simple bounds on the function C_O. First of all, since our observers should be non-intrusive, we can show that the SCG of S is a sub graph of the SCG of $S \parallel O$, and therefore $C_O(S) \geq 1$. Also, in the case of the **leadsto** pattern, the transitions and places-based observers add exactly one place to the net associated to S. In this case, we can show that the complexity of these two observers is always less than 2; we can at most double the size of the system. We can prove a similar upper bound for the **leadsto** observer based on data. While the three observers have the same (theoretical) worst-case complexity, our experiments have shown that one approach was superior to the others. We are not aware of previous work on using experimental criteria to select the best observer for a real time property. In the context of "untimed properties", this approach may be compared to the problem of optimising the generation of Bchi Automata from LTL formulas, see e.g. Gastin & Oddoux (2001).

We have used our prototype compiler to experiment with different implementations for the observers. The goal is to find the most efficient observer "in practice", that is the observer with the lowest complexity. To this end, we have used a fixed set of representative examples and for a specific set of properties (we consider both valid and invalid properties). Fig. 4 shows a synthesis of the results obtained for the **leadsto** pattern. In this paper, we consider only three case examples, for they exhibit very different features (size of the state space, amount of concurrency and symmetry in the

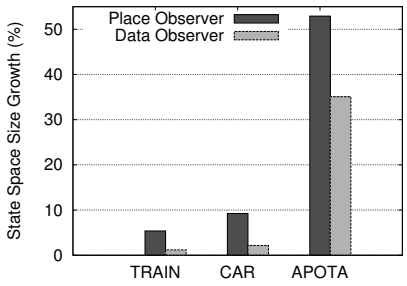


Figure 4: Compared complexity of the data and place observers (in percentage of system size growth) for invalid properties (above) and valid properties (below).

system, ...):

- TRAIN is a model of a train gate controller. The example models a system responsible for controlling the barriers protecting a railroad crossing gate. When a train approaches, the barrier must be lowered and then raised after the train's departure. The valid property, for the TRAIN example, states that the delay between raising and lowering a barrier does not exceed 100 units of time. For the invalid property, we use the same requirement, but shortening the delay to 75.

- APOTA is an industrial use case that models the network protocol in charge of data communications between an air plane and ground stations, Berthomieu et al. (2010*b*). This example has been obtained using a translation from AADL to Fiacre. In this case, timing constraints arise from timeouts between requests and periods of the tasks involved in the protocol implementation. The property, in this case, is related to the worst-case execution time for the main application task.

- CAR is a system modelling an automated rail car system taken from Dong et al. (2008). The system is composed of four terminals connected by rail tracks in a cyclic network. Several rail cars, operated from a central control centre, are available to transport passengers between terminals. When a car approaches its destination, it sends a request to the terminal to signal its arrival. Passengers in the terminal can then book a travel in the car. The valid property, for the CAR example, states that a passenger arriving in a terminal, must have a car ready to transport him within 15 units of time. For the invalid property, we use the same requirement, but shortening the delay to 2 units of time.

In Fig. 4, we compare the growth in the state space size—that is the value of $C_o(S)$—for the place and data observers defined in Section. 4.1 and our three running examples. We do not consider the transition observer in these results since the events used in the requirements are all related to a system entering a state (and therefore our benchmark favor the place observer over the transition observer).

4.3 Proving the correctness of TTS observers

We start by giving sufficient conditions for an observer O to be *non-intrusive*, meaning that the observer does not interfere with the observed system. Formally, we show that any trace σ of the observed system N is preserved in the composed system $N \parallel O$: the observer does not obstruct a behaviour of the system (see Lemma 1 below). Conversely, we show that, from any trace of the composition $N \parallel O$, we can obtain a trace of N by erasing the events from O: the observer does not add new behaviours to the system. This is actually a consequence of Property 1.

Let $\Sigma(N)$ be the set of well-formed traces of the TTS N. We write T_{sync} the set of synchronised transitions of the observer, that is, the set of transitions t of O such that $L(t) = L(t')$ for some transitions t' of N. We define T_{imm} as the set of transitions of the observer whose static time interval is $[0, 0]$. By construction, no transiton in T_{sync} can also be part of T_{imm}.

Lemma 1 *Assume O satisfies the following conditions:*

- *all synchronised transitions have a trivial static time interval and no priority (that is, for every t in T_{sync}, $I_s^t = [0; +\infty[$ and t has no priority over another transition in O);*
- *from any state of the observer, and for every label $l \in L_{sync}$, there is at least one transition t in O with label l that can fire immediately;*
- *from any state of the observer, there is no infinite sequence of transitions in T_{imm}.*

then, for all timed trace σ in $\Sigma(N)$ there exists a timed trace σ' in $\Sigma(N \parallel O)$ and a trace σ_o in $\Sigma(O)$ such that $\sigma \bowtie \sigma_o \sim \sigma'$.

That is, every trace σ in N still appears in the composed system $N \parallel O$.

Proof sketch: Given a trace σ in $\Sigma(N)$, we build a trace σ_o in $\Sigma(O)$ that is composable with σ. This is done by induction on the size of finite prefixes of σ (in accordance with Def. 9). The last condition ensures that the observer does not introduce an infinite number of interruptions within a finite time interval (that is, σ_o is well-formed).

The conditions in Lemma 1 are true for the **leadsto** observer O_d defined in Fig. 3. Therefore this observer cannot interfere with the system under observation. Next, we prove that the transition observer is sound, meaning that it reports correctly if its associated pattern is valid or not. We prove the soundness of this observer by showing that, for any TTS N, the event error does not appear in the traces of $N \parallel O_d$ if and only if the pattern is valid for N. We write $\mathsf{error} \in N \parallel O_d$ to mean that there exists a trace σ' in $\Sigma(N \parallel O_d)$ such that $\mathsf{error} \in \sigma'$.

Theorem 1 *We have $\mathsf{error} \notin N \parallel O_d$ if and only if, for all $\sigma \in \Sigma(N)$ such that $\sigma = \sigma^a(E_1, \tau_1)\sigma^b$, there exist σ^c and σ^d with $\sigma^b = \sigma^c(E_2, \tau_2)\sigma^d$, $E_2 \notin \sigma^c$, and $\tau_2 - \tau_1 \leqslant max$.*

Proof

This is a consequence of the two following properties, in which we assume that $\sigma_1 \bowtie \sigma_2 \sim \sigma'$ holds, with $\sigma_1 \in \Sigma(N)$ and $\sigma_2 \in \Sigma(O_d)$.

Property 2 *If there exist* σ_1^a, σ_1^b, *and* σ_1^c *such that* $\sigma_1 = \sigma_1^a(E_1, \tau_1)\sigma_1^b\sigma_1^c \wedge \mathsf{sup}(d(\sigma_1^b)) - \tau_1 \geqslant max \wedge E_2 \notin \sigma_1^b$, *then* $\mathsf{error} \in \sigma_2$.

Proof By Definitions 8 and 9, and since $\sigma_1 \bowtie \sigma_2 \sim \sigma'$ holds, $E_1 \in \sigma_1 \Rightarrow E_1 \in \sigma_2$ and $\exists \sigma_2^b$, a finite prefix of σ_2, such that $\sigma_1^b \bowtie \sigma_2^b$ holds. By hypothesis, $\mathsf{sup}(d(\sigma_1^b)) - \tau_1 \geqslant max$ and $E_2 \notin \sigma_1^b$. Thus, $\mathsf{sup}(d(\sigma_2^b)) - \tau_1 \geqslant max$ and $E_2 \notin \sigma_2^b$. In σ_2^b, transition error is enabled for duration max, and is therefore fired in σ_2, that is $\mathsf{error} \in \sigma_2$.

Property 3 *If* $\mathsf{error} \in \sigma_2$, *then there exist* σ_1^a, σ_1^b, *and* σ_1^c *such that* $\sigma_1 = \sigma_1^a(E_1, \tau_1)\sigma_1^b\sigma_1^c \wedge \mathsf{sup}(d(\sigma_1^b)) - \tau_1 \geqslant max \wedge E_2 \notin \sigma_1^b$.

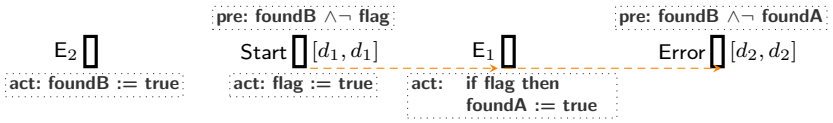
Proof By hypothesis, $\mathsf{error} \in \sigma_2$. By construction of O_d, this means that **flag** was true for max units of time, that is, σ_2 is of the form $\sigma_2^a(E_1, \tau_1)\sigma_2^b(error, \tau_2)\sigma_2^c$ with $\tau_2 - \tau_1 = max$ and $E_2 \notin \sigma_2^b$. As a consequence of Def. 8, σ_1 is of the form $\sigma_1^a(E_1, \tau_1)\sigma_1^b\sigma_1^c$, with $E_2 \notin \sigma_1^b$ and $\mathsf{sup}(d(\sigma_1^b)) - \tau_1 \geqslant max$.

5 Catalog of observers

We introduce, for each pattern, the corresponding TTS observer, as well as the associated LTL formula. By convention, Error, Start, ... are transitions that belong to the observer, whereas E_1 (resp. E_2) stand for all the transitions of the system that match predicate A (resp. B). We also use the symbol I as a shorthand for the time interval $[d_1, d_2]$. The observers for the pattern obtained with other time intervals–such as $]d_1, d_2]$, $]d_1, +\infty[$, or in the case $d_1 = d_2$–are essentially the same, except for some priorities between transitions that may change. By convention, the boolean variables used in the definition of an observers are initially set to false.

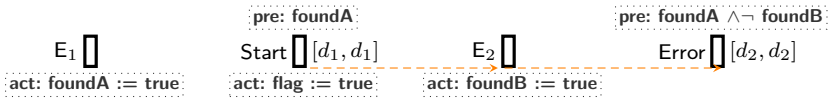
5.1 Simple observers

- **Present** A **after** B **within** $[d_1, d_2]$: The LTL formula to verify is $[]\neg$Error.



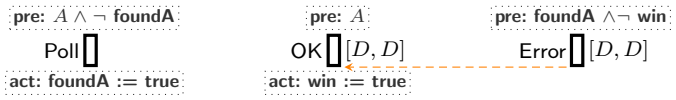
In this observer, transition **Error** is conditioned by the value of the shared boolean variables **foundA** and **foundB**. Variable **foundB** is set to true after transition E_2 and transition **Error** is enabled only if the predicate **foundB** $\wedge\neg$ **foundA** is true. Transition **Start** is fired d_1 u.t after an occurrence of E_2 (because it is enabled when **foundB** is true and **flag** is false). Then, after the first occurrence of E_1 and if **flag** is true, **foundA** is set to true. This captures the first occurrence of E_1 after **Start** has been fired. After d_2 u.t., in the absence of E_1, transition **Error** is fired. Therefore, the verification of the pattern boils down to checking if the event **Error** is reachable. The priority (dashed arrows) between **Start**, **Error**, and E_1 is here necessary to ensure that occurrences of E_1 occurring at d_1 or d_2 are taken into account.

- **Present** A **before** B **within** I : The LTL formula to verify is $(\Diamond B) \Rightarrow \neg\Diamond(\mathsf{Error} \vee (\mathsf{foundB} \wedge \neg\mathsf{flag}))$.



  Like in the previous case, variables **foundA** and **foundB** are used to record the occurrence of transitions E_1 and E_2. Transition **Start** is fired, and variable **flag** is set to true, d_1 u.t. after the first E_1. Then transition **Error** is fired only if its precondition—the predicate **foundA** $\wedge\neg$ **foundB**—is true for d_2 u.t. Therefore transition **Error** is fired if and only if there is an occurrence of E_2 before E_1 (because then **foundB** is true) or if the first occurrence of E_2 is not within $[d_1, d_2]$ of the first occurrence of E_1.

- **Present** A **lasting** D : The LTL formula to verify is $\Box\neg\mathsf{Error}$.

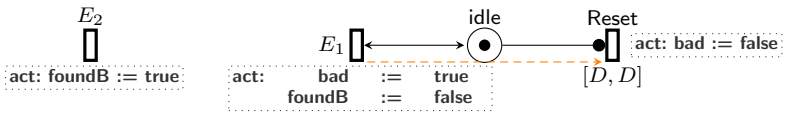


  Variable **foundA** is set to true when transition $Poll$ is fired, that is when **A** becomes true for the first time. Transition **OK** is used to set **win** to true if **A** is true for duration D without interruption (otherwise its timing constraint is resetted). Otherwise, if variable **win** is still false after D u.t., then transition $Error$ is fired. We use a priority between $Error$ and OK to disambiguate the behaviour D u.t. after **Poll** is fired.

- **Absent** A **after** B **for interval** I : The LTL formula to verify is $\Diamond B \Rightarrow \Diamond\mathsf{Error}$.

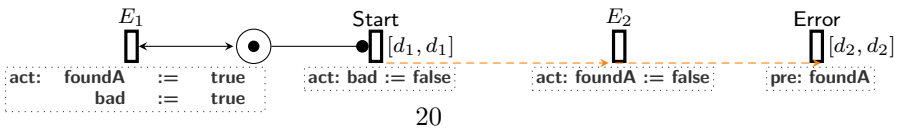
  We use the same observer as for *Present* A *after* B *within* I, but here Error is the expected behaviour.

- **Absent** A **before** B **for duration** D : The LTL formula to verify is $\Box\neg(\mathsf{foundB} \wedge \mathsf{bad})$.



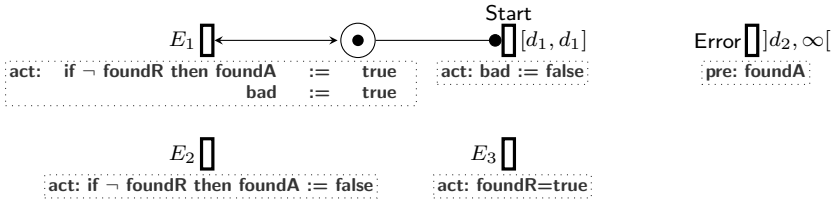
  Variable $foundB$ is set to true after each occurrence of E_2. Conversely, we set the variables *bad* to true and $foundB$ to false at each occurrence of E_1. Therefore $foundB$ is true on every "time interval" between an E_2 and an E_1. We use transition *Reset* to set *bad* to false if this interval is longer than D. As a consequence, the pattern holds if we cannot find an occurrence of E_2 ($foundB$ is true) while *bad* is true.

- A **leadsto** B **within** I : The LTL formula to verify is $(\Box\neg\mathsf{Error}) \wedge (\Box\neg(B \wedge \mathsf{bad}))$.



After each occurrence of E_1, variables $foundA$ and bad are set to true and the transition $Start$ is enabled. Variable bad is used to control the beginning of the time interval. After each occurrence of E_2 variable $foundA$ is set to false. Hence $Error$ is fired if there is an occurrence of E_1 not followed by an occurrence of E_2 after d_2 u.t. We use priorities to avoid errors when E_2 occurs precisely at time d_1 or d_2.

- A **leadsto** B **within** I **before** R : The LTL formula to verify is $\Diamond R \Rightarrow (\Box\neg\mathsf{Error} \wedge \Box\neg(B \wedge \mathsf{bad}))$.



  Same explanation than for the previous case, but we only take into account transitions E_1 and E_2 occurring before E_3.

- A **leadsto** B **within** I **after** R : The LTL formula to verify is $\Diamond R \Rightarrow (\Box\neg\mathsf{Error} \wedge \Box\neg(B \wedge \mathsf{bad}))$.

  It is similar to the observer of the pattern A **leadsto** first B **within** I **before** R . We should just replace $\neg$foundR in transition E_1 and E_2 by foundR.

  Same explanation than in the previous case, but we only take into account transitions E_1 and E_2 occurring after an E_3.

5.2 Observers for Composed patterns

To check a composed pattern, we use a combination of the respective observers, as well as a combination of the respective LTL formulas. For instance, if (T_1, ϕ_1) and (T_2, ϕ_2) are the observers and LTL formulas corresponding to the patterns P_1 and P_2, then the composite pattern P_1 *and* P_2 is checked using the LTL formula $\phi_1 \wedge \phi_2$. Similarly, if we check the LTL formula $\phi_1 \Rightarrow \phi_2$ (implication), we obtain a composite pattern $P_1 \multimap P_2$ that is satisfied by systems T such that, for all traces of T, the pattern P_2 holds whenever P_1 holds.

6 Use cases and experimental results

In this section, we report on three experiments that have been performed using an extension of a Fiacre compiler that automatically compose a system with the necessary observers. In case the system does not meet its specification, we obtain a counter-example that can be converted into a timed sequence of events exhibiting a problematic scenario. This sequence can be played back using two programs provided in the TINA tool set, *nd* and *play*. The first program is a graphical animator for Time Petri Net, while the latter is an interactive (text-based) animator for the full TTS model.

Avionic Protocol and AADL. Our first example is a network avionic protocol (NPL) which includes several functions allowing the pilot and ground stations to receive and send information relative to the plane: weather, speed, ... AADL has been used to model the dynamic architecture for this demonstrator, Berthomieu et al. (2010*a*). The AADL model includes several threads that exchange information through shared memory data and amounts to about 8 diagrams and 800 lines of code (using AADL textual syntax). The AADL code specifies both the hardware and software architecture of the system and defines the real time properties of threads, like for instance their dispatch protocol (periodic or sporadic) or their periods.

We used the AADL2Fiacre plug-in of Topcased to check properties on the NPL specification. The Fiacre model obtained after transformation takes into account the complete behavior described in the AADL model but also the whole language execution model, meaning that our interpretation takes fully into account the scheduling semantics as specified in the AADL standard. The abstract state space for the TTS generated from Fiacre has about 120 000 states and 180 000 transitions and can be generated in less than 12s on a typical development computer (Intel dual-core processor at 2GHz with 2Gb of RAM). On examples of this size, our model checker is able to prove formal properties in a few seconds. We checked a set of 22 requirements that were given together with the description of the system, all expressed using a natural language description and, in one case, a scenario based on a UML sequence diagram. Of these 22 requirements, 18 where instances of "untimed patterns", such as checking the absence of deadlock or that threads are resettable. The four remaining requirements where "response patterns" of the kind A **leadsto** first B **within** $[0, d]$. Using patterns, we were able to check the 22 patterns in less than 5min.

Service Oriented Applications. We consider models obtained from the composition of services expressed using a timed extension of BPEL, the Business Process Execution Language. Our example models a scenario from the health-care domain related to patient handling during a medical examination. The scenario involves three entities, each one managed by a service: a Clinic Service (CS); a Medical Analysis Service (MAS); and a Pharmacy Service (PS). When a patient arrives in clinic, a doctor should check with the MCS whether its social security number is valid. If so, the doctor may order some medical analyses from the MAS and, after analyzing the results, he can order drugs through the PS. Timing constraints can be added to this scenario by associating a duration to each activity of the workflow and a delay to each service invocation.

We use our patterns to express different requirements on this system. An example involving the absence pattern is that we cannot have two medical analyses for a patient in less than 10 days (240 hours): **absent** MAS.medicalAnalysis **after** MAS.medicalAnalysis **for interval** $]0, 240]$. A more complicated example of requirement is to impose that if a doctor does not cancel a drug order within 6 hours, then it should not cancel drugs for another 48 hours. This requirement can be expressed using the composition of two absence patterns (see Section. 3.4):

(**absent** MCS.drugsChanging **after** MCS.drugsAsking **for interval** $[0; 6]$)
$\multimap$ (**absent** MCS.drugsChanging **after** MCS.drugsAsking **for interval** $[0; 54]$).

Finally, using the notation S.init and S.end to refer to a start (resp. end) event in the

service S, we can express that drugs must be delivered within 48 hours of the medical examination start: MCS.init **leadsto** PS.sendDrugsOrder **within** $[0; 48]$.

The complete scenario is given in Guermouche & Dal Zilio (2012), where we describe a transformation tool chain from Timed BPEL processes to Fiacre. For a more complex version of the health care scenario, with seven different services and more concurrent activities, the state graph for the TTS generated from Fiacre is quite small, with only 886 states and 2476 transitions. The generation of the Fiacre specification and its corresponding state space takes less than a second. For examples of this size, the verification time for checking a requirement is negligible (half a second).

Transportation Systems. Our third example is an automated railcar system, taken from Dong et al. (2008), that was directly modeled using Fiacre. It is composed of four terminals connected by rail tracks in a cyclic network. Several railcars, operated from a central control center, are available to transport passengers between terminals. When a car approaches its destination, it sends a request to signal its arrival to the terminal. This system has several real-time constraints: the terminal must be ready to accommodate an incoming car in 5s; a car arriving in a terminal leaves its door open for exactly 10s; passengers entering a car have 5s to choose their destination; etc. There are three key requirements:

(P1) when a passenger arrives in a terminal, a car must be ready to transport him within 15s. This property can be expressed with a response pattern, where **Passenger/sndReq** is the state where the passenger requests a car and **Car/ackTerm** is the state where it is served:

$$\textbf{Passenger/sendReq}\ \textit{leadsto}\ \textbf{Car/ackTerm}\ \textit{within}\,[0, 15]$$

(P2) When the car starts moving, the door must be closed:

$$\textit{present}\ \textbf{CarDoor/closeDoor}\ \textit{after}\ \textbf{CarHandler/moving}\ \textit{within}\,[0, 10]$$

(P3) When a passenger selects a destination (in the car), a signal should stay illuminated until the car has arrived:

$$\textit{absent}\ \textbf{Terminal/buttonOff}\ \textit{before}\ \textbf{Control/ackTerm}\ \textit{for duration}\,10$$

These three patterns are valid on our Fiacre model. Concerning performance, we generate the complete state space in 310ms, using 400kB of memory. This gives an upper-bound to the complexity of checking simple (untimed) reachability properties on the system, like for instance the absence of deadlocks. The three patterns can all be checked in less than 1.5s. For instance, we observed that checking property (P1) is not more complex than exploring the complete system: the property is checked in 450ms, using 780kB of memory. Also, this is roughly the same complexity than checking the corresponding untimed requirement in LTL that is: $\Box$(**Passenger/sendReq** $\Rightarrow$ $\Diamond$**Control/ackTerm**).

Conclusion. According to several benchmarks, it appears that the complexity of checking timed patterns is in the same order of magnitude than checking their untimed temporal logic equivalent. An exception to this observation is when the temporal values used in the patterns are far different from those found in the system; for example if

checking a periodic system, with a period of a few milliseconds, against a requirement using an interval of a few minutes. These experiments, while still modest in size, give a good appraisal of the use of formal verification techniques for real industrial software.

7 Contributions and perspectives

We have reduce the problem of checking real-time properties on a given model to the problem of checking LTL properties on the composition of this model with an observer. We have also defined a real-time pattern language based on the work of Dwyer et al. (1999) and inspired from real-case studies. To choose the best way to verify a pattern, we have defined, for each pattern, a set of non-intrusive observers. We have proposed a formal framework to prove the correctness of observers, in particular regarding their non-interfererence with the system under observation.

Our approach has been integrated into a complete verification tool chain for the Fiacre modelling language. Experimental results have helped us designing the most efficient observers. Indeed, another contribution of our work is the use of a pragmatic approach for comparing the effectiveness of different observers for a given property. In this context, data observers look promising.

We are following several directions for future work. A first goal is to define a new low-level language for observers—adapted from the TTS model—equipped with more powerful optimisation techniques and with easier soundness proofs. On the theoretical side, we are currently looking into the use of mechanised theorem proving techniques to support the validation of observers. On the experimental side, we need to define an improved method to select the best observer. For instance, we would like to provide a tool for the "syntax-directed selection" of observers that would choose (and even adapt) the right observers based on a structural analysis of the target system.

8 References

References

Abid, N., Dal Zilio, S. & Le Botlan, D. (2012), A Verified Approach for Checking Real-Time Specification Patterns, *in* 'VECOS 2012'.

Aceto, L., Bouyer, P., Burgueño, A. & Larsen, K. G. (2003), 'The power of reachability testing for timed automata', *Theoretical Computer Science* **300**(1-3), 411–475.

Aceto, L., Burgueño, A. & Larsen, K. G. (1998), Model checking via reachability testing for timed automata, *in* 'TACAS'98', Vol. 1384 of *LNCS*, Springer, pp. 263–280.

Bayse, E., Cavalli, A., Núñez, M. & Zaïdi, F. (2005), 'A passive testing approach based on invariants: application to the wap', *Comput. Netw.* **48**(2), 247–266.

Behrmann, G., David, A. & Larsen, K. G. (2004), A tutorial on uppaal, *in* M. Bernardo & F. Corradini, eds, 'SFM', Vol. 3185 of *LNCS*, Springer, pp. 200–236.

Berthomieu, B. (2012), 'The Fiacre Compiler', `http://projects.laas.fr/fiacre`.

Berthomieu, B., Bodeveix, J.-P., Dal Zilio, S., Dissaux, P., Filali, M., Gaufillet, P., Heim, S. & Vernadat, F. (2010*a*), Formal Verification of AADL models with Fiacre and Tina, *in* 'ERTSS 2010 - Embedded Real-Time Software and Systems', pp. 1–9.

Berthomieu, B., Bodeveix, J.-P., Dal Zilio, S., Dissaux, P., Filali, M., Heim, S., Gaufillet, P. & Vernadat, F. (2010*b*), Formal Verification of AADL models with Fiacre and Tina, *in* 'ERTSS 2010 – Embedded Real-Time Software and Systems'.

Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F. & Vernadat, F. (2008), Fiacre: an Intermediate Language for Model Verification in the Topcased Environment, *in* 'ERTS 2008'.

Berthomieu, B., Ribet, P.-O. & Vernadat, F. (2004), 'The tool TINA – Construction of Abstract State Spaces for Time Petri Nets', *Int. Journal of Production Research*.

Bianculli, D., Ghezzi, C., Pautasso, C. & Senti, P. (2012), Specification patterns from research to industry: a case study in service-based applications, *in* 'ICSE 2012', IEEE Computer Society Press, pp. 992–1000.

Dong, J. S., Hao, P., Qin, S., Sun, J. & Yi, W. (2008), 'Timed automata patterns', *IEEE Trans. Software Eng.* **34**(6), 844–859.

Dwyer, M. B., Avrunin, G. S. & Corbett, J. C. (1999), Patterns in property specifications for finite-state verification, *in* 'ICSE'99', pp. 411–420.

Esterel Technologies (n.d.), 'Scade tool suite'.

Gastin, P. & Oddoux, D. (2001), Fast ltl to büchi automata translation, *in* 'CAV'01', Springer, pp. 53–65.

Gruhn, V. & Laue, R. (2006), 'Patterns for timed property specifications', *Electr. Notes Theor. Comput. Sci.* **153**(2), 117–133.

Guermouche, N. & Dal Zilio, S. (2012), Towards Timed Requirement Verification for Service Choreographies, *in* '8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing', p. 10.

Henzinger, T. A. (1998), It's about time: Real-time logics reviewed, *in* D. Sangiorgi & R. de Simone, eds, 'CONCUR', Vol. 1466 of *LNCS*, Springer, pp. 439–454.

Konrad, S. & Cheng, B. H. C. (2005), Real-time specification patterns, *in* G.-C. Roman, W. G. Griswold & B. Nuseibeh, eds, 'ICSE', ACM, pp. 372–381.

Koymans, R. (1990), 'Specifying real-time properties with metric temporal logic', *Real-Time Syst.* **2**(4), 255–299.

Merlin, P. (1974), *A Study of the Recoverability of Computing Systems*, University of California, Irvine.

Ouaknine, J. & Worrell, J. (2007), On the decidability and complexity of metric temporal logic over finite words, *in* 'Logical Methods in Computer Science', p. 2007.

Peres, F., Berthomieu, B. & Vernadat, F. (2011), 'On the composition of time petri nets', *Discrete Event Dynamic Systems* **21**(3), 395–424.

Schimpf, A., Merz, S. & Smaus, J.-G. (2009), Construction of bchi automata for ltl model checking verified in isabelle/hol, *in* 'TPHOLs '09', Springer-Verlag, pp. 424–439.

Toussaint, J., Simonot-Lion, F. & Thomesse, J.-P. (1997), Time constraints verification methods based on time petri nets, *in* 'FTDCS', IEEE Computer Society, pp. 262–269.

Appendices

A TTS Composition

Here is the full definition of TTS composition (only sketched in Definition 7).

Definition 10 (Composition of two TTS) *Assuming N_1 and N_2 are defined as above, let N be the TTS corresponding to their composition, which we write $N = N_1 \parallel N_2$. It is defined by the 10-tuple $(P, T, B, F, M^0, I^s, \mathcal{L}, L, S, <)$ where:*

1. $P = P_1 \cup P_2$
2. *Let $\perp$ be an element not in $T_1 \cup T_2$. Let $T_1^\perp$ be $T_1 \cup \{\perp\}$ and $T_2^\perp$ be $T_2 \cup \{\perp\}$. We define T as the following subset of $T_1^\perp \times T_2^\perp$:*
$$\begin{array}{rcl} T & = & \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, L_1(t_1) = L_2(t_2)\} \\ & \cup & \{(t_1, \perp) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronised}\} \\ & \cup & \{(\perp, t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronised}\} \end{array}$$
3. *The backward incidence function B is defined as follows:*
$$\begin{array}{rcl} B(\,(t_1, t_2)\,) & = & B_1(t_1) + B_2(t_2) \text{ when } L_1(t_1) = L_2(\\ B(\,(t_1, \perp)\,) & = & B_1(t_1) \text{ when } t_1 \text{ is not synchronised} \\ B(\,(\perp, t_2)\,) & = & B_2(t_2) \text{ when } t_2 \text{ is not synchronised} \end{array}$$
4. *The forward incidence function is defined similarly.*
5. *M^0 equals M_1^0 on P_1 and equals M_2^0 on P_2.*
6. *The function I^s is defined as follow:*
$$\begin{array}{rcl} I^s(\,(t_1, t_2)\,) & = & [0, +\infty[\text{ when } L_1(t_1) = L_2(t_2) \\ I^s(\,(t_1, \perp)\,) & = & I_1^s(t_1) \text{ when } t_1 \text{ is not synchronised.} \\ I^s(\,(\perp, t_2)\,) & = & I_2^s(t_2) \text{ when } t_2 \text{ is not synchronised.} \end{array}$$
7. $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$
8. $L : T \to \mathcal{L} \cup \{\varepsilon\}$ *such that:*
$$\begin{array}{rcl} L(t) & = & L(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, L(t_1) = L(t_2) \\ & \cup & L(t_1) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronised} \\ & \cup & L(t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronised} \end{array}$$
9. $S = S_1 \times S_2$

10. *$<_1$ is a binary relation on T_1. We may freely consider it as a binary relation on $T_1^\perp$ (and so there is no $t \in T_1^\perp$ with $t < \perp$ or $\perp < t$). Similarly, $<_2$ is considered as a binary relation on $T_2^\perp$. Then, $<$ is defined by: for all $(t_1, t_1', t_2, t_2') \in (T_1^\perp)^2 \times (T_2^\perp)^2$, we have $(t_1, t_2) < (t_1', t_2')$ if and only if $t_1 < t_1'$ or $t_2 < t_2'$. As required, $<$ is transitive (the proof uses Def. 6).*

Additionally, the initial state is defined as $((M_1^0, M_2^0), (s_1^0, s_2^0), (I_1^s, I_2^s))$.